Speculation-aware Cluster Scheduling

Xiaoqi Ren[1], Ganesh Ananthanarayanan[2], Adam Wierman[1], Minlan Yu[3]
[1]California Institute of Technology, [2]Microsoft Research, [3]University of Southern California
{xren,adamw} @caltech.edu, ga@microsoft.com, minlanyu@usc.edu

ABSTRACT

Stragglers are a crucial roadblock to achieving predictable performance in today's clusters. Speculation has been widely-adopted in order to mitigate the impact of stragglers; however speculation mechanisms are designed and operated independently of job scheduling when, in fact, scheduling a speculative copy of a task has a direct impact on the resources available for other jobs. In this work, based on a simple model and its analysis, we design Hopper, a job scheduler that is speculation-aware, i.e., that integrates the tradeoffs associated with speculation into job scheduling decisions.

1. INTRODUCTION

As the scale and complexity of clusters increase, hard-to-model systemic interactions that degrade the performance of tasks become common [2, 7]. Consequently, many tasks become "stragglers", i.e., running slower than expected, leading to significant unpredictability (and delay) in job completion times – tasks in Facebook's Hadoop cluster can run up to 8× slower than expected [2]. The most successful and widely deployed straggler mitigation solution is *speculation*, i.e., speculatively running extra copies of tasks that have already, or are likely to become, stragglers, and then picking the earliest of the copies to finish, e.g., [2, 4, 5, 8, 14]. Speculation is commonplace in production clusters, e.g., in our analysis of Facebook's Hadoop cluster speculative tasks account for 25% of all tasks and 21% of resource usage.

Speculation is unavoidably intertwined with job scheduling because spawning a speculative copy of a task has a direct impact on the resources available for *other* jobs. Aggressive speculation can improve the performance of the job at hand while hurting the performance of other jobs. Despite this, speculation policies deployed today are all designed and operated *independently* of job scheduling; schedulers simply allocate slots to speculative copies in a "best-effort" fashion, e.g., [2, 4, 5, 8, 10, 14].

Coordinating speculation and scheduling decisions is an opportunity for significant performance improvement. In this paper, we formulate an analytical model to capture the interaction between speculation within jobs and resource allocation across jobs to optimize job completion times. Based on analysis of the model and statistic characterizations of production traces [2–4, 9], we propose the *first speculation-aware job scheduler*, Hopper, which dynamically allocates slots to jobs keeping in mind the speculation requirements necessary for predictable performance. Hopper incorporates a variety of factors such as fairness, dependencies (DAGs) between tasks, etc. Further, Hopper is compatible with all current speculation algorithms.

The key insight behind Hopper is that a scheduler must anticipate the speculation requirements of jobs and dynamically allocate capacity depending on the marginal value (in terms of performance) of extra slots, which are likely used for speculation. A novel observation that leads to the design of Hopper is that there is a sharp threshold in the marginal value of extra slots – an extra slot is always more beneficial for a job below its threshold than it is for any job above its threshold. The identification of this threshold then allows Hopper to use different resource allocation strategies depending on whether the system capacity is such that all jobs can be allocated more slots than their threshold or not. This leads to a dynamic, adaptive, online scheduler that reacts to the current system load in a manner that appropriately weighs the value of speculation.

We have built two demonstration prototypes by augmenting the scheduling frameworks Hadoop [1] (for batch jobs) and Spark [12] (for interactive jobs) and show that 50% improvements over state-of-the-art schedulers and speculation strategies can be achieved through the coordination of scheduling and speculation. These are reported on in [13], without any discussion of the theory behind the design. Here we describe the model formulation and analysis that led to the design of Hopper.

2. MODEL OVERVIEW

We focus on a system with S slots, each of which can have one task scheduled to it. Jobs arrive over time and the ith arrival is denoted by J_i and has T_i tasks, each of which has an i.i.d. random task completion time τ. We denote the remaining number of tasks for the ith job at time t by $T_i(t)$.

The key piece of our model is the characterization of the service rate of the ith job, $\mu_i(t)$, as a function how many slots, S_i, it is allocated and the average number of speculative copies per task at time t, $k(t)$. Note that $\mu_i(t)$ should be interpreted as the completion rate of the ith job. We adopt the following approximation for $\mu_i(t)$, which has been used previously in the design of task level speculation policies by [4].

$$\mu_i(t) = \min\left(S_i, T_i(t)k(t)\right) \times \left(\frac{E[\tau]}{k(t)E\left[\min(\tau_1, \ldots, \tau_{k(t)})\right]}\right) \quad (1)$$

To understand this approximate model, note that the first term approximates the number of slots the job occupied and the second term approximates the "blow up factor," i.e., the ratio of the expected work completed without speculative copies to the amount of work done with speculative copies. To understand the first term, note that there

are $T_i(t)k(t)$ tasks available to schedule at time t, including speculative copies. Given that the maximum capacity that can be allocated is S_i, we obtain the first term in (1). The second term is the the expected amount of work done per task without speculation ($E[\tau]$) divided by the expected amount of work done per task with speculation ($k(t)E[\min(\tau_1, \tau_2, \ldots, \tau_{k(t)})]$), since $k(t)$ copies are created and then they are stopped when the first copy completes.

To specialize (1) further, we note that task completion times often show evidence of Pareto tails [2–4, 9]. So, we focus on the case of Pareto(x_m,β) completion times. Given this form for the task completion time distributions, the optimal speculation level has been shown in [4] to be as in (2).

$$k(t) = \begin{cases} \frac{2}{\beta}, & S_i \leq \frac{2}{\beta}T_i(t) \\ S_i/T_i(t), & S_i > \frac{2}{\beta}T_i(t); \end{cases} \quad (2)$$

Plugging the optimal speculation level given in (2) into the model for $\mu_i(t)$ in (1) yields the following model for the service rate.

$$\mu_i(t) = \begin{cases} \frac{\beta^2}{4(\beta-1)}S_i(t), & S_i \leq \frac{2}{\beta}T_i(t) \\ \frac{\beta}{\beta-1}T_i(t) - \frac{1}{\beta-1}\frac{T_i^2(t)}{S_i}, & S_i > \frac{2}{\beta}T_i(t); \end{cases} \quad (3)$$

Equation 3 surprisingly shows that the marginal return of an extra slot has a sharp threshold (when $S_i = \frac{2}{\beta}T_i(t)$), where below the threshold, the marginal return is large and above the threshold, the marginal return is small. So it is desirable to ensure that every job is allocated enough slots to reach the threshold (if possible) before giving any job slots beyond this threshold. Thus, we refer to this threshold as the "desired (minimum) allocation" for a job or simply the "virtual job size" as following:

$$V_i(t) = \frac{2}{\beta}T_i(t) \quad (4)$$

3. HOPPER

There are two distinct cases one must consider for resources allocating across jobs: (i) How should slots be allocated if there are not enough slots to give every job to perform optimal speculation? (ii) How should slots be allocated if there are more than enough slots to give every job to perform optimal speculation.

When the system is *capacity constrained*, our analytic results highlight that the job scheduler should give as many jobs as possible their desired allocation, i.e., their full virtual job sizes. Thus, the scheduler should start with the job with the smallest virtual job size $V_i(t)$ and work its way to larger jobs giving all the jobs the optimal level until capacity is exhausted.

However, when the system is *not* capacity constrained, then the key design challenge becomes how to divide the extra capacity among the jobs present. Our analytic results highlight that the job scheduler should do a form of *proportional sharing* to determine the allocation of slots to jobs. Specifically, jobs should be allocated slots proportionally to their virtual job sizes, i.e., job i receives

$$\left(\frac{V_i(t)}{\sum_j V_j(t)}\right) S = \left(\frac{T_i(t)}{\sum_j T_j(t)}\right) S \text{ slots}, \quad (5)$$

where S is the number of slots available in the system. In the above we have assumed $V_i(t) = (2/\beta)T_i(t)$, as discussed above.

Concretely, we prove that the following algorithm is completion rate optimal.

Algorithm 1 (HOPPER, SINGLE-PHASED).
Let $J(t) = \{J_1, J_2, \ldots, J_n\}$ denote the jobs in the system at time t sorted in increasing order of remaining tasks, so $T_1(t) \leq \ldots \leq T_n(t)$.

1. *If $S \leq \frac{2}{\beta}\sum T_i(t)$, then assign $S_i = \frac{2}{\beta}T_i(t)$ to jobs in order from $i = 1$ to n until no slots remain and assign $S_i = 0$ for all remaining jobs.*
2. *If $S > \frac{2}{\beta}\sum T_i(t)$, then assign $S_i = \left(\frac{T_i(t)}{\sum T_j(t)}\right) S$ for all jobs $J_i \in J(t)$.*

THEOREM 1. *Algorithm 1 is completion rate maximal for single-phased jobs, i.e., it maximizes $\sum \mu_i(t)$.*

In a real system implementation, once a slot becomes available, Hopper chooses a job and assigns the vacant slot to it immediately to avoid unnecessary resource waste. In that case, Hopper tries best to approximate the ideal allocation i.e., it will assign the slot to job i if $i = \text{argmax}_{i \in J(t)} H_i - S_i$, where H_i is the number of slots that the job should have based on Algorithm 1, and S_i is the number of slots the job currently has. See [13] for details and for comments on implementation in a distributed setting.

3.1 Incorporating Fairness

Fairness is an important constraint on cluster scheduling. To allow some flexibility, while still tightly controlling the unfairness introduced, instead of allocating equal number of slots to every job, we define a notion of *approximate* fairness as follows. We say that a scheduler is ϵ-fair if it guarantees that every job receives at least $S/N(t)(1-\epsilon)$ slots at *any* time t, where $N(t)$ is the number of jobs in the system at time t. $\epsilon \to 0$ indicates perfect fairness while $\epsilon \to 1$ indicate focusing on performance.

Hopper can be extended to guarantee ϵ-fairness while maintaining optimality as follows.

Algorithm 2 (HOPPER, FAIRNESS).
Let $J(t) = \{J_1, J_2, \ldots, J_n\}$ denote the jobs in the system at time t sorted in increasing order of remaining tasks, so $T_1(t) \leq \ldots \leq T_n(t)$. Define m_1 such that $i \leq m_1$ implies $\frac{2}{\beta}T_i(t) \leq \frac{S}{N} - \epsilon$.

1. *If $S \leq \frac{2}{\beta}\sum_{i=m_1+1}^{n} T_i(t) + m_1\frac{S}{N}(1-\epsilon)$, begin by assigning all jobs $\frac{S}{N}(1-\epsilon)$ slots. Then assign an additional $\frac{2}{\beta}T_i(t) - \frac{S}{N}(1-\epsilon)$ slots to jobs J_i from $i = m_1 + 1$ to n until no slots remain.*
2. *If $S > \frac{2}{\beta}\sum_{i=m_1+1}^{n} T_i(t) + m_1\frac{S}{N}(1-\epsilon)$, then define m_2 as the minimum value such that*

$$\frac{T_{m_2+1}(t)}{\sum_{i=m_2+1}^{N} T_i(t)}(S - m_2\frac{S}{N}(1-\epsilon)) \geq \max\{\frac{S}{N}(1-\epsilon), \frac{2}{\beta}T_{m_2+1}(t)\}.$$

Then, assign $\frac{S}{N}(1-\epsilon)$ slots to jobs J_i with $1 \leq i \leq m_2$, and assign $\frac{T_i(t)}{\sum_{i=m_2+1}^{N} T_i(t)}\left(S - m_2\frac{S}{N}(1-\epsilon)\right)$ slots to jobs J_i with $m_2 + 1 \leq i \leq N$.

THEOREM 2. *Algorithm 2 is completion rate maximal among ϵ-fair allocations.*

3.2 Heterogeneous Job DAGs

We can also extend our analysis of single-phased jobs to jobs with multi-phased DAGs of tasks that have varied communication patterns (e.g., many-to-one or all-to-all). We consider multiple phases that are not separated by strict barriers but are rather *pipelined*. Downstream tasks do not wait for *all* the upstream tasks to finish but read the upstream outputs as the tasks finish.

The scheduler's goal is to balance the gains due to overlapping network utilization while still favoring upstream phases with smaller number of tasks. We capture this using a simple weighting factor, α per job, set to be the ratio of remaining work in network transfer in the downstream phase to the work in the upstream phase. The definition of α makes our scheduler favors jobs with higher remaining communication and lower remaining tasks in the running phase.

Given the weighting factor α, our analytic results highlight that the structural form of Algorithm 1 do not change. However, the following adjustments are required.

First, the prioritization of jobs based on $T_i(t)$ should be replaced by a prioritization of jobs based on $\max\{T_i(t), T_i'(t)\}$, where $T_i(t)$ is the remaining number of tasks in the current phase and $T_i'(t)$ is the remaining work in communication in the downstream phase. This adjustment is motivated by the work of [11], which proves that, so-called, MaxSRPT is 2-speed optimal for completion times.[1] However, the model in [11] does not include stragglers, and so we need to supplement MaxSRPT.

Also, we redefine the virtual size of a job to include α as

$$V_i(t) = \frac{2}{\beta} T_i(t)\sqrt{\alpha_i}.$$

This change means that capacity is shared as follows: job i receives

$$\left(\frac{V_i(t)}{\sum V_j(t)}\right) S = \left(\frac{T_i(t)\sqrt{\alpha_i}}{\sum T_j(t)\sqrt{\alpha_j}}\right) S \text{ slots.} \qquad (6)$$

We use the weighting factor α_i to understand how to adjust the desired allocation of jobs depending on the relative sizes of job i in the current phase and the following phase. For example, by setting $\alpha_i = T_i'/T_i$, it captures number of tasks created in the next phase per task completed in the current phase, which is appropriate when adjacent phases in the DAG can be pipelined. Mathematically, one can show that if we seek to maximize the α-weighted throughput, i.e. $\sum_i \alpha_i \mu_i(t)$, then the desired allocation changes from $(2/\beta)T_i(t)$ to $(2/\beta)T_i(t)\sqrt{\alpha_i/\alpha_{min}}$, where α_{min} is the smallest α_j among the jobs that are currently running. This leads to the following algorithm for the case of DAGs of tasks.

Algorithm 3 (HOPPER, DAGS OF TASKS).
Let $J(t) = \{J_1, J_2, \ldots, J_n\}$ denote the jobs in the system at time t sorted in ascending order of $\max\{T_i(t), T_i'(t)\}$. If J_i and J_j have the same $\max\{T_i(t), T_i'(t)\}$ then the job with larger weight is listed first. Let $\alpha_{\min}^{(k)}$ denote the minimum weight of weights for first k jobs in $J(t)$, so $\alpha_{\min}^{(k)} = \min\{\alpha_1, \alpha_2, \ldots, \alpha_k\}$. And let $J_{k_{\min}}$ denote the job which has the minimum weight in first k jobs, so the weight of $J_{k_{\min}}$ is $\alpha_{\min}^{(k)}$. Let $V_i(t)$ denote the virtual size for job $J_i \in J(t)$, so $V_i(t) = \frac{2}{\beta} T_i(t)\sqrt{\alpha_i}$.

1. *If $S \leq \frac{V_1(t)}{\sqrt{\alpha_{\min}^{(2)}}}$, assign $S_1 = S$ and $S_i = 0$ for $i > 1$.*
2. *If $\exists\, k < n$ such that $\sum_{i=1}^{k} \frac{V_i(t)}{\sqrt{\alpha_{\min}^{(k+1)}}} < S \leq \sum_{i=1}^{k+1} \frac{V_i(t)}{\sqrt{\alpha_{\min}^{(k+1)}}}$, assign $S_i = \frac{V_i(t)}{\sqrt{\alpha_{\min}^{(k+1)}}}$ for i in order of $\{1, 2, \ldots, k_{\min}-1, k_{\min}+1, \ldots, k, k+1, k_{\min}\}$ until no slots remain, and $S_i = 0$ for $i > k+1$.*
3. *If $\exists\, k < n-1$ such that $\sum_{i=1}^{k+1} \frac{V_i(t)}{\sqrt{\alpha_{\min}^{(k+1)}}} < S \leq \sum_{i=1}^{k+1} \frac{V_i(t)}{\sqrt{\alpha_{\min}^{(k+2)}}}$, then assign $S_i = \frac{V_i(t)}{\sum_{i=1}^{k+1} V_i(t)} S$ for $i = 1, \ldots, k+1$, and $S_i = 0$ for $i > k+1$.*
4. *If $\sum_{i=1}^{n} \frac{V_i(t)}{\sqrt{\alpha_{\min}^{(n)}}} < S$, then assign $S_i = \frac{V_i(t)}{\sum_{i=1}^{n} V_i(t)} S$ for $i = 1, 2, \ldots, n$.*

[1] 2-speed optimal means that MaxSRPT guarantees completion time better than the optimal in the original system, if it is given twice the service capacity. Note that [11] also shows that it is impossible to be constant-competitive without being granted extra service capacity.

4. REFERENCES

[1] Hadoop. `http://hadoop.apache.org`.

[2] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *USENIX NSDI*, 2013.

[3] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *USENIX NSDI*, 2012.

[4] G. Ananthanarayanan, M. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. GRASS: Trimming Stragglers in Approximation Analytics. In *USENIX NSDI*, 2014.

[5] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, E. Harris, and B. Saha. Reining in the Outliers in Map-Reduce Clusters Using Mantri. In *USENIX OSDI*, 2010.

[6] H. Chen, J. Marden, and A. Wierman. On the Impact of Heterogeneity and Back-end Scheduling in Load Balancing Designs. In *INFOCOM*. IEEE, 2009.

[7] J. Dean and L. Barroso. The Tail at Scale. *Communications of the ACM*, (2), 2013.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 2008.

[9] F. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized Task-aware Scheduling for Data Center Networks. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 431–442. ACM, 2014.

[10] O. K, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[11] M. Lin, L. Zhang, A. Wierman, and J. Tan. Joint Optimization of Overlapping Phases in MapReduce. *Performance Evaluation*, 2013.

[12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *USENIX NSDI*, 2012.

[13] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu. Hopper: Decentralized Speculation-aware Cluster Scheduling at Scale. *ACM SIGCOMM*, 2015.

[14] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *USENIX OSDI*, 2008.